wiki.archlinux.org

GnuPG - ArchWiki

43-55 minutes

According to the official website:

> GnuPG is a complete and free implementation of the OpenPGP standard as defined by RFC 4880 (also known as PGP). GnuPG allows you to encrypt and sign your data and communications; it features a versatile key management system, along with access modules for all kinds of public key directories. GnuPG, also known as GPG, is a command line tool with features for easy integration with other applications. A wealth of frontend applications and libraries are available. GnuPG also provides support for S/MIME and Secure Shell (ssh).

Installation

Install the gnupg package.

This will also install pinentry, a collection of simple PIN or passphrase entry dialogs which GnuPG uses for passphrase entry. The shell script `/usr/bin/pinentry` determines which *pinentry* dialog is used, in the order described at #pinentry.

If you want to use a graphical frontend or program that integrates with GnuPG, see List of applications/Security#Encryption, signing, steganography.

Configuration

Home directory

The GnuPG home directory is where the GnuPG suite stores its keyrings and private keys, and reads configurations from. By default, the path used is `~/.gnupg`. There are two ways to override this:

- Set the `$GNUPGHOME` environment variable.
- Use the `--homedir` argument, e.g. `$ gpg --homedir path/to/file` [1].

By default, the home directory has its permissions set to `700` and the files it contains have their permissions set to `600`. Only the owner of the directory has permission to read, write, and access the files. This is for security purposes and should not be changed. In case this directory or any file inside it does not follow this security measure, you will get warnings about unsafe file and home directory permissions.

Configuration files

All of GnuPG's behavior is configurable via command line arguments. For arguments you would like to be the default, you can add them to the respective configuration file:

- *gpg* checks *gnupg_home*`/gpg.conf` (user) and `/etc/gnupg/gpg.conf` (global) [2]. Since *gpg* is the main entrypoint for GnuPG, most configuration of interest will be here. See GPG Options for possible options.
- *dirmngr* checks *gnupg_home*`/dirmngr.conf` and `/etc/gnupg/dirmngr.conf`. *dirmngr* is a program internally

invoked by `gpg` to access PGP keyservers [3]. See Dirmngr Options for possible options.

These two configuration files cover the common usecases, but there are more auxiliary programs in the GnuPG suite with their own options. See the GnuPG manual for a comprehensive list.

Create the desired file(s), and set their permissions to `600` as discussed in #Home directory.

Add to these files any long options you want. Do not write the two dashes, but simply the name of the option and required arguments. For example, to make GnuPG always use a keyring at a specific path, as if it was invoked as `gpg --no-default-keyring --keyring keyring-path ...`:

gnupg_home/gpg.conf (or /etc/gnupg/gpg.conf)

no-default-keyring
keyring *keyring-path*

Other examples are found in #See also.

Additionally, pacman uses a different set of configuration files for package signature verification. See Pacman/Package signing for details.

Default options for new users

If you want to setup some default options for new users, put configuration files in `/etc/skel/.gnupg/`. When the new user is added in system, files from here will be copied to its GnuPG home directory. There is also a simple script called *addgnupghome* which you can use to create new GnuPG home directories for existing users:

addgnupghome user1 user2

This will add the respective `/home/user1/.gnupg/` and `/home/user2/.gnupg/` and copy the files from the skeleton directory to it. Users with existing GnuPG home directory are simply skipped.

Usage

Note:

- Whenever a *`user-id`* is required in a command, it can be specified with your key ID, fingerprint, a part of your name or email address, etc. GnuPG is flexible on this.
- Whenever a *`key-id`* is needed, it can be found adding the `--keyid-format=long` flag to the command. To show the master secret key for example, run `gpg --list-secret-keys --keyid-format=long` *`user-id`*, the *key-id* is the hexadecimal hash provided on the same line as *sec*.

Create a key pair

Generate a key pair by typing in a terminal:

$ gpg --full-gen-key

Also add the `--expert` option to the command line to access more ciphers and in particular the newer ECC cipher (Wikipedia:Elliptic-curve cryptography).

The command will prompt for answers to several questions. For general use most people will want:

- The default *RSA and RSA* for sign and encrypt keys.
- A keysize of the default 3072 value. A larger keysize of 4096 "gives us almost nothing, while costing us quite a lot" (see why doesn't GnuPG default to using RSA-4096).

- An expiration date: a period of one year is good enough for the average user. This way even if access is lost to the keyring, it will allow others to know that it is no longer valid. At a later stage, if necessary, the expiration date can be extended without having to re-issue a new key.
- Your name and email address. You can add multiple identities to the same key later (*e.g.*, if you have multiple email addresses you want to associate with this key).
- *no* optional comment. Since the semantics of the comment field are not well-defined, it has limited value for identification.
- A secure passphrase, find some guidelines in Security#Choosing secure passwords.

  Note: The name and email address you enter here will be seen by anybody who imports your key.

  Tip: The simpler `--gen-key` option uses default parameters for the key cipher, size and expiry and only asks for *real name* and *email address*.

  List keys

  To list keys in your public key ring:

  $ gpg --list-keys

  To list keys in your secret key ring:

  $ gpg --list-secret-keys

  Export your public key

  GnuPG's main usage is to ensure confidentiality of exchanged messages via public-key cryptography. With it each user distributes the public key of their keyring, which can be used by

others to encrypt messages to the user. The private key must *always* be kept private, otherwise confidentiality is broken. See Wikipedia:Public-key cryptography for examples about the message exchange.

So, in order for others to send encrypted messages to you, they need your public key.

To generate an ASCII version of a user's public key to file *`public-key`*`.asc` (e.g. to distribute it by e-mail):

$ gpg --export --armor --output *public-key*.asc *user-id*

Alternatively, or in addition, you can #Use a keyserver to share your key.

Tip:

- Add `--no-emit-version` to avoid printing the version number, or add the corresponding setting to your configuration file.
- You can omit the `user-id` to export all public keys within your keyring. This is useful if you want to share multiple identities at once, or for importing in another application, e.g. Thunderbird.

Import a public key

In order to encrypt messages to others, as well as verify their signatures, you need their public key. To import a public key with file name *`public.key`* to your public key ring:

$ gpg --import *public-key*.asc

Alternatively, #Use a keyserver to find a public key.

If you wish to import a key ID to install a specific Arch Linux package, see pacman/Package signing#Managing the keyring and Makepkg#Signature checking.

Use a keyserver

Sending keys

You can register your key with a public PGP key server, so that others can retrieve it without having to contact you directly:

$ gpg --send-keys *key-id*

Warning: Once a key has been submitted to a keyserver, it cannot be deleted from the server. The reason is explained in the MIT PGP Public Key Server FAQ.

Note: The associated email address, once published publicly, could be the target of spammers and in this case anti-spam filtering may be necessary.

Searching and receiving keys

To find out details of a key on the keyserver, without importing it, do:

$ gpg --search-keys *user-id*

To import a key from a key server:

$ gpg --recv-keys *key-id*

Warning:

- You should verify the authenticity of the retrieved public key by comparing its fingerprint with one that the owner published on an independent source(s) (e.g., contacting the person directly). See Wikipedia:Public key fingerprint for more information.
- It is recommended to use the long key ID or the full fingerprint when receiving a key. Using a short ID may encounter collisions. All keys will be imported that have the short ID, see fake keys

found in the wild for such example.

Tip: Adding `auto-key-retrieve` to the GPG configuration file will automatically fetch keys from the key server as needed. This is not a compromise on security, but it can be considered a **privacy violation**; see "web bug" in gpg(1).

Key servers

The most common keyservers are:

- Ubuntu Keyserver: federated, no verification, keys cannot be deleted.
- Mailvelope Keyserver: central, verification of email IDs, keys can be deleted.
- keys.openpgp.org: central, verification of email IDs, keys can be deleted, no third-party signatures (i.e. no Web of Trust support).

More are listed at Wikipedia:Key server (cryptographic)#Keyserver examples.

An alternative key server can be specified with the `keyserver` option in one of the #Configuration files, for instance:

~/.gnupg/dirmngr.conf

keyserver hkp://keyserver.ubuntu.com

A temporary use of another server is handy when the regular one does not work as it should. It can be achieved by, for example,

$ gpg --keyserver *hkps://keys.openpgp.org/* --search-keys *user-id*

Tip:

- If receiving fails with the message `gpg: keyserver receive`

`failed: General error`, and you use the default hkps keyserver pool, make sure set the HKPS pool verification certificate with `hkp-cacert /usr/share/gnupg/sks-keyservers.netCA.pem` in your `dirmngr.conf` and kill the old dirmngr process.

- If receiving fails with the message `gpg: keyserver receive failed: Connection refused`, try using a different DNS server.
- You can connect to the keyserver over Tor with Tor#Torsocks. Or using the `--use-tor` command line option. See [4] for more information.
- You can connect to a keyserver using a proxy by setting the `http_proxy` environment variable and setting `honor-http-proxy` in `dirmngr.conf`. Alternatively, set `http-proxy` *`host[:port]`* in the configuration file to override the environment variable of the same name. Restart the `dirmngr.service` user service for the changes to take effect.

Web Key Directory

The Web Key Service (WKS) protocol is a new standard for key distribution, where the email domain provides its own key server called Web Key Directory (WKD). When encrypting to an email address (e.g. `user@example.com`), GnuPG (>=2.1.16) will query the domain (`example.com`) via HTTPS for the public OpenPGP key if it is not already in the local keyring. The option `auto-key-locate` will locate a key using the WKD protocol if there is no key on the local keyring for this email address.

gpg --recipient *user@example.org* --auto-key-locate --encrypt *doc*

See the GnuPG Wiki for a list of email providers that support WKD. If you control the domain of your email address yourself, you can follow this guide to enable WKD for your domain. To check if your key can be found in the WKD you can use this webinterface.

Encrypt and decrypt

Asymmetric

You need to #Import a public key of a user before encrypting (option `-e`/`--encrypt`) a file or message to that recipient (option `-r`/`--recipient`). Additionally you need to #Create a key pair if you have not already done so.

To encrypt a file with the name *doc*, use:

$ gpg --recipient *user-id* --encrypt *doc*

To decrypt (option `-d`/`--decrypt`) a file with the name *doc*.gpg encrypted with your public key, use:

$ gpg --output *doc* --decrypt *doc*.gpg

gpg will prompt you for your passphrase and then decrypt and write the data from *doc*.gpg to *doc*. If you omit the `-o`/`--output` option, *gpg* will write the decrypted data to stdout.

Tip:

- Add `--armor` to encrypt a file using ASCII armor, suitable for copying and pasting a message in text format.
- Use `-R` *`user-id`* or `--hidden-recipient` *`user-id`* instead of `-r` to not put the recipient key IDs in the encrypted message. This helps to hide the receivers of the message and is a limited countermeasure against traffic analysis.

- Add `--no-emit-version` to avoid printing the version number, or add the corresponding setting to your configuration file.
- You can use GnuPG to encrypt your sensitive documents by using your own user-id as recipient or by using the `--default-recipient-self` flag; however, you can only do this one file at a time, although you can always tarball various files and then encrypt the tarball. See also Data-at-rest encryption#Available methods if you want to encrypt directories or a whole file-system.

  Symmetric

  Symmetric encryption does not require the generation of a key pair and can be used to simply encrypt data with a passphrase. Simply use `-c`/`--symmetric` to perform symmetric encryption:

  $ gpg -c *doc*

  The following example:

- Encrypts *doc* with a symmetric cipher using a passphrase
- Uses the AES-256 cipher algorithm to encrypt the data
- Uses the SHA-512 digest algorithm to mangle the passphrase and generate the encryption key
- Mangles the passphrase for 65536 iterations

  $ gpg -c --s2k-cipher-algo AES256 --s2k-digest-algo SHA512 --s2k-count 65536 *doc*

  To decrypt a symmetrically encrypted *doc*`.gpg` using a passphrase and output decrypted contents into the same directory as *doc* do:

$ gpg --output *doc* --decrypt *doc*.gpg

Directory

Encrypting/decrypting a directory can be done with gpgtar(1).

Encrypt:

$ gpgtar -c -o *dir*.gpg *dir*

Decrypt:

$ gpgtar -d *dir*.gpg

Key maintenance

Backup your private key

To backup your private key do the following:

$ gpg --export-secret-keys --armor --output *private-key*.asc *user-id*

Note the above command will require that you enter the passphrase for the key. This is because otherwise anyone who gains access to the above exported file would be able to encrypt and sign documents as if they were you *without* needing to know your passphrase.

Warning:

- The passphrase is usually the weakest link in protecting your secret key. Place the private key in a safe place on a different system/device, such as a locked container or encrypted drive. It is the only safety you have to regain control to your keyring in case of, for example, a drive failure, theft or worse.
- This method of backing up key has some security limitations. See https://web.archive.org/web/20210803213236/https:

//habd.as/post/moving-gpg-keys-privately/ for a more secure way to back up and import key using *gpg*.

To import the backup of your private key:

$ gpg --import *private-key*.asc

Tip: Paperkey can be used to export private keys as human readable text or machine readable barcodes that can be printed on paper and archived.

Backup your revocation certificate

Revocation certificates are automatically generated for newly generated keys. These are by default located in `~/.gnupg/openpgp-revocs.d/`. The filename of the certificate is the fingerprint of the key it will revoke. The revocation certificates can also be generated manually by the user later using:

$ gpg --gen-revoke --armor --output *revcert*.asc *user-id*

This certificate can be used to #Revoke a key if it is ever lost or compromised. The backup will be useful if you have no longer access to the secret key and are therefore not able to generate a new revocation certificate with the above command. It is short enough to be printed out and typed in by hand if necessary.

Warning: Anyone with access to the revocation certificate can revoke the key publicly, this action cannot be undone. Protect your revocation certificate like you protect your secret key.

Edit your key

Running the `gpg --edit-key *user-id*` command will present a menu which enables you to do most of your key management related tasks.

Type `help` in the edit key sub menu to show the complete list of commands. Some useful ones:

> passwd # change the passphrase
> clean # compact any user ID that is no longer usable (e.g revoked or expired)
> revkey # revoke a key
> addkey # add a subkey to this key
> expire # change the key expiration time
> adduid # add additional names, comments, and email addresses
> addphoto # add photo to key (must be JPG, 240x288 recommended, enter full path to image when prompted)

Tip: If you have multiple email accounts you can add each one of them as an identity, using `adduid` command. You can then set your favourite one as `primary`.

Exporting subkey

If you plan to use the same key across multiple devices, you may want to strip out your master key and only keep the bare minimum encryption subkey on less secure systems.

First, find out which subkey you want to export.

$ gpg --list-secret-keys --with-subkey-fingerprint

Select only that subkey to export.

$ gpg -a --export-secret-subkeys [subkey id]! > /tmp/subkey.gpg

Warning: If you forget to add the !, all of your subkeys will be exported.

At this point you could stop, but it is most likely a good idea to change the passphrase as well. Import the key into a temporary folder.

```
$ gpg --homedir /tmp/gpg --import /tmp/subkey.gpg
$ gpg --homedir /tmp/gpg --edit-key user-id
> passwd
> save
$ gpg --homedir /tmp/gpg -a --export-secret-subkeys [subkey id]!
> /tmp/subkey.altpass.gpg
```

Note: You will get a warning that the master key was not available and the password was not changed, but that can safely be ignored as the subkey password was.

At this point, you can now use `/tmp/subkey.altpass.gpg` on your other devices.

Extending expiration date

Warning: Never delete your expired or revoked subkeys unless you have a good reason. Doing so will cause you to lose the ability to decrypt files encrypted with the old subkey. Please **only** delete expired or revoked keys from other users to clean your keyring.

It is good practice to set an expiration date on your subkeys, so that if you lose access to the key (e.g. you forget the passphrase) the key will not continue to be used indefinitely by others. When the key expires, it is relatively straight-forward to extend the expiration date:

```
$ gpg --edit-key user-id
> expire
```

You will be prompted for a new expiration date, as well as the passphrase for your secret key, which is used to sign the new expiration date.

Repeat this for any further subkeys that have expired:

> key 1
> expire

Finally, save the changes and quit:

> save

Update it to a keyserver.

$ gpg --keyserver keyserver.ubuntu.com --send-keys *key-id*

Alternatively, if you use this key on multiple computers, you can export the public key (with new signed expiration dates) and import it on those machines:

$ gpg --export --output pubkey.gpg *user-id*
$ gpg --import pubkey.gpg

There is no need to re-export your secret key or update your backups: the master secret key itself never expires, and the signature of the expiration date left on the public key and subkeys is all that is needed.

Rotating subkeys

Warning: Never delete your expired or revoked subkeys unless you have a good reason. Doing so will cause you to lose the ability to decrypt files encrypted with the old subkey. Please **only** delete expired or revoked keys from other users to clean your keyring.

Alternatively, if you prefer to stop using subkeys entirely once they have expired, you can create new ones. Do this a few weeks in advance to allow others to update their keyring.

Tip: You do not need to create a new key simply because it is expired. You can extend the expiration date, see the section #Extending expiration date.

Create new subkey (repeat for both signing and encrypting key)

$ gpg --edit-key *user-id*
> addkey

And answer the following questions it asks (see #Create a key pair for suggested settings).

Save changes

> save

Update it to a keyserver.

$ gpg --keyserver *pgp.mit.edu* --send-keys *user-id*

You will also need to export a fresh copy of your secret keys for backup purposes. See the section #Backup your private key for details on how to do this.

Tip: Revoking expired subkeys is unnecessary and arguably bad form. If you are constantly revoking keys, it may cause others to lack confidence in you.

Revoke a key

Key revocation should be performed if the key is compromised, superseded, no longer used, or you forget your passphrase. This is done by merging the key with the revocation certificate of the key.

If you have no longer access to your keypair, first #Import a public key to import your own key. Then, to revoke the key, import the file saved in #Backup your revocation certificate:

$ gpg --import *revcert*.asc

Now the revocation needs to be made public. #Use a keyserver to send the revoked key to a public PGP server if you used one in the past, otherwise, export the revoked key to a file and

distribute it to your communication partners.

Signatures

Signatures certify and timestamp documents. If the document is modified, verification of the signature will fail. Unlike encryption which uses public keys to encrypt a document, signatures are created with the user's private key. The recipient of a signed document then verifies the signature using the sender's public key.

Create a signature

Sign a file

To sign a file use the `-s`/`--sign` flag:

$ gpg --output *doc*.sig --sign *doc*

doc`.sig` contains both the compressed content of the original file *doc* and the signature in a binary format, but the file is not encrypted. However, you can combine signing with encrypting.

Clearsign a file or message

To sign a file without compressing it into binary format use:

$ gpg --output *doc*.sig --clearsign *doc*

Here both the content of the original file *doc* and the signature are stored in human-readable form in *doc*`.sig`.

Make a detached signature

To create a separate signature file to be distributed separately from the document or file itself, use the `--detach-sig` flag:

$ gpg --output *doc*.sig --detach-sig *doc*

Here the signature is stored in *doc*`.sig`, but the contents of *doc* are not stored in it. This method is often used in distributing software projects to allow users to verify that the program has not been modified by a third party.

Verify a signature

To verify a signature use the `--verify` flag:

$ gpg --verify *doc*.sig

where *doc*`.sig` is the signed file containing the signature you wish to verify.

If you are verifying a detached signature, both the signed data file and the signature file must be present when verifying. For example, to verify Arch Linux's latest iso you would do:

$ gpg --verify archlinux-*version*.iso.sig

where `archlinux-`*version*`.iso` must be located in the same directory.

You can also specify the signed data file with a second argument:

$ gpg --verify archlinux-*version*.iso.sig */path/to/*archlinux-*version*.iso

If a file has been encrypted in addition to being signed, simply decrypt the file and its signature will also be verified.

gpg-agent

gpg-agent is mostly used as daemon to request and cache the password for the keychain. This is useful if GnuPG is used from

an external program like a mail client. gnupg comes with systemd user sockets which are enabled by default. These sockets are `gpg-agent.socket`, `gpg-agent-extra.socket`, `gpg-agent-browser.socket`, `gpg-agent-ssh.socket`, and `dirmngr.socket`.

- The main `gpg-agent.socket` is used by *gpg* to connect to the *gpg-agent* daemon.
- The intended use for the `gpg-agent-extra.socket` on a local system is to set up a Unix domain socket forwarding from a remote system. This enables to use *gpg* on the remote system without exposing the private keys to the remote system. See gpg-agent(1) for details.
- The `gpg-agent-browser.socket` allows web browsers to access the *gpg-agent* daemon.
- The `gpg-agent-ssh.socket` can be used by SSH to cache SSH keys added by the *ssh-add* program. See #SSH agent for the necessary configuration.
- The `dirmngr.socket` starts a GnuPG daemon handling connections to keyservers.

  Note: If you use non-default GnuPG #Home directory, you will need to edit all socket files to use the values of `gpgconf --list-dirs`. The socket names use the hash of the non-default GnuPG home directory [5], so you can hardcode it without worrying about it changing.

  Configuration

  gpg-agent can be configured via `~/.gnupg/gpg-agent.conf` file. The configuration options are listed in gpg-agent(1). For

example you can change cache ttl for unused keys:

~/.gnupg/gpg-agent.conf

default-cache-ttl 3600

Tip: To cache your passphrase for the whole session, please run the following command:

$ /usr/lib/gnupg/gpg-preset-passphrase --preset XXXXX

where XXXXX is the keygrip. You can get its value when running `gpg --with-keygrip -K`. The passphrase will be stored until `gpg-agent` is restarted. If you set up `default-cache-ttl` value, it will take precedence.

Reload the agent

After changing the configuration, reload the agent using *gpg-connect-agent*:

$ gpg-connect-agent reloadagent /bye

The command should print 0K.

However in some cases only the restart may not be sufficient, like when `keep-screen` has been added to the agent configuration. In this case you firstly need to kill the ongoing gpg-agent process and then you can restart it as was explained above.

pinentry

`gpg-agent` can be configured via the `pinentry-program` stanza to use a particular pinentry user interface when prompting the user for a passphrase. For example:

~/.gnupg/gpg-agent.conf

pinentry-program /usr/bin/pinentry-curses

There are other pinentry programs that you can choose from - see `pacman -Ql pinentry | grep /usr/bin/`.

Tip:

- In order to use `/usr/bin/pinentry-kwallet` you have to install the kwalletcli[AUR] package.
- `/usr/bin/pinentry-gtk-2` and `/usr/bin/pinentry-gnome3` support the DBus Secret Service API, which allows for remembering passwords via a compliant manager such as GNOME Keyring or KeePassXC.

  Remember to reload the agent after making changes to the configuration.

  Cache passwords

  `max-cache-ttl` and `default-cache-ttl` defines how many seconds gpg-agent should cache the passwords. To enter a password once a session, set them to something very high, for instance:

  gpg-agent.conf

  max-cache-ttl 60480000
  default-cache-ttl 60480000

  For password caching in SSH emulation mode, set `default-cache-ttl-ssh` and `max-cache-ttl-ssh` instead, for example:

  gpg-agent.conf

  default-cache-ttl-ssh 60480000
  max-cache-ttl-ssh 60480000

Unattended passphrase

Starting with GnuPG 2.1.0 the use of gpg-agent and pinentry is required, which may break backwards compatibility for passphrases piped in from STDIN using the `--passphrase-fd 0` commandline option. In order to have the same type of functionality as the older releases two things must be done:

First, edit the gpg-agent configuration to allow *loopback* pinentry mode:

~/.gnupg/gpg-agent.conf

allow-loopback-pinentry

Reload the agent if it is running to let the change take effect.

Second, either the application needs to be updated to include a commandline parameter to use loopback mode like so:

$ gpg --pinentry-mode loopback ...

...or if this is not possible, add the option to the configuration:

~/.gnupg/gpg.conf

pinentry-mode loopback

Note: The upstream author indicates setting `pinentry-mode loopback` in `gpg.conf` may break other usage, using the commandline option should be preferred if at all possible. [6]

SSH agent

gpg-agent has OpenSSH agent emulation. If you already use the GnuPG suite, you might consider using its agent to also cache your SSH keys. Additionally, some users may prefer the PIN entry dialog GnuPG agent provides as part of its passphrase management.

Set SSH_AUTH_SOCK

Set the following variables to communicate with *gpg-agent* instead of the default *ssh-agent*.

```
SSH_AGENT_PID=""
SSH_AUTH_SOCK="${XDG_RUNTIME_DIR}/gnupg/S.gpg-agent.ssh"
```

Note:

- If you are using a script to manage your variables, you may also unset `SSH_AGENT_PID` rather than setting it to `""`, via `unset SSH_AGENT_PID`.
- If you set your `SSH_AUTH_SOCK` manually, keep in mind that your socket location may be different if you are using a custom `GNUPGHOME`. You can use the following bash example, or change `SSH_AUTH_SOCK` to the value of `gpgconf --list-dirs agent-ssh-socket`.
- If GNOME Keyring is installed, it is necessary to deactivate its ssh component. Otherwise, it will overwrite `SSH_AUTH_SOCK`.

  Alternatively, depend on Bash. This works for non-standard socket locations as well:

  ~/.bashrc

  ```
  unset SSH_AGENT_PID
  if [ "${gnupg_SSH_AUTH_SOCK_by:-0}" -ne $$ ]; then
    export SSH_AUTH_SOCK="$(gpgconf --list-dirs agent-ssh-socket)"
  fi
  ```

  Note: The test involving the `gnupg_SSH_AUTH_SOCK_by` variable is for the case where the agent is started as `gpg-`

`agent --daemon /bin/sh`, in which case the shell inherits the `SSH_AUTH_SOCK` variable from the parent, *gpg-agent* [7].

Configure pinentry to use the correct TTY

Also set the GPG_TTY and refresh the TTY in case user has switched into an X session as stated in gpg-agent(1). For example:

~/.bashrc

```
export GPG_TTY=$(tty)
gpg-connect-agent updatestartuptty /bye >/dev/null
```

If you use multiple terminals simultaneously and want *gpg-agent* to ask for passphrase via *pinentry-curses* from the same terminal where the *ssh* command was run, add the following to the SSH configuration file. This will make the TTY to be refreshed every time an *ssh* command is run [8]:

~/.ssh/config

```
Match host * exec "gpg-connect-agent UPDATESTARTUPTTY /bye"
```

Note that GPG_TTY environment variable has to be set for this to work.

Add SSH keys

Once *gpg-agent* is running you can use *ssh-add* to approve keys, following the same steps as for ssh-agent. The list of approved keys is stored in the `~/.gnupg/sshcontrol` file.

Once your key is approved, you will get a *pinentry* dialog every time your passphrase is needed. For password caching see

#Cache passwords.

Using a PGP key for SSH authentication

You can also use your PGP key as an SSH key. This requires a key with the `Authentication` capability (see #Custom capabilities). There are various benefits gained by using a PGP key for SSH authentication, including:

- Reduced key maintenance, as you will no longer need to maintain an SSH key.
- The ability to store the authentication key on a smartcard. GnuPG will automatically detect the key when the card is available, and add it to the agent (check with `ssh-add -l` or `ssh-add -L`). The comment for the key should be something like: `openpgp:`*`key-id`* or `cardno:`*`card-id`*.

  To retrieve the public key part of your GPG/SSH key, run `gpg --export-ssh-key` *`gpg-key`*. If your key is authentication-capable but this command still fails with "Unusable public key", add a `!` suffix ([9]).

  Unless you have your GPG key on a keycard, you need to add your key to `$GNUPGHOME/sshcontrol` to be recognized as a SSH key. If your key is on a keycard, its keygrip is added to `sshcontrol` implicitly. If not, get the keygrip of your key this way:

  $ gpg --list-keys --with-keygrip

  sub rsa4096 2018-07-25 [A]
  Keygrip =
  1531C8084D16DC4C36911F1585AF0ACE7AAFD7E7

  Then edit `sshcontrol` like this. Adding the keygrip is a one-

time action; you will not need to edit the file again, unless you are adding additional keys.

$GNUPGHOME/sshcontrol

1531C8084D16DC4C36911F1585AF0ACE7AAFD7E7

Forwarding gpg-agent and ssh-agent to remote

It is possible to forward one's gpg-agent to a remote machine by forwarding gpg sockets to the remote machine, as explained by the GnuPG wiki.

First, add the following line to `/etc/ssh/sshd_config` on the remote machine to enable automatic removal of stale sockets on connect. Without this, the socket(s) on the remote machine will need to removed manually before connecting with forwarding enabled for agent forwarding to work:

/etc/ssh/sshd_config

...
StreamLocalBindUnlink yes
...

Note: You will have to invoke `systemctl reload sshd` on the remote machine for the new configuration to be loaded by sshd.

On the client, use the `RemoteForward` SSH directive to forward traffic destined for a remote port, to a port on your local host. As described in ssh_config(5) § RemoteForward, this directive's parameters are the listening socket path on the remote, and then the destination socket path on the local host. Your configuration should look something like this:

$HOME/.ssh/config

Host *remote_name*

...

RemoteForward *remote_agent_socket* *local_agent_extra_socket*

RemoteForward *remote_agent_ssh_socket* *local_agent_ssh_socket*

The first line configures gpg-agent forwarding:

- *remote_agent_socket* is the output of `gpgconf --list-dir agent-socket` on the remote host.
- *local_agent_extra_socket* is `gpgconf --list-dir agent-extra-socket` on the local host.

The second line is optional. It configures ssh-agent forwarding:

- *local_agent_ssh_socket* is `gpgconf --list-dir agent-ssh-socket` on the remote host.
- *remote_agent_ssh_socket* is `gpgconf --list-dir agent-ssh-socket` on the local host.

Note: If using ssh-agent forwarding, the remote should have `SSH_AUTH_SOCK` set to the output of `gpgconf --list-dir agent-ssh-socket` as mentioned in #SSH agent).

So, with the default paths, it would be:

RemoteForward /run/user/1000/gnupg/S.gpg-agent /run/user/1000/gnupg/S.gpg-agent.extra

RemoteForward /run/user/1000/gnupg/S.gpg-agent.ssh /run/user/1000/gnupg/S.gpg-agent.ssh

With this configuration in place, invoking `ssh myremote` should automatically forward the gpg-agent to the remote, and allow the

use of your gpg key(s) for both decryption/signing (and allows the use of ssh-agent with gpg if the second `RemoteForward` line is included).

Smartcards

GnuPG uses *scdaemon* as an interface to your smartcard reader, please refer to the man page scdaemon(1) for details.

GnuPG only setups

Note: To allow scdaemon direct access to USB smartcard readers the optional dependency libusb-compat must be installed

If you do not plan to use other cards but those based on GnuPG, you should check the `reader-port` parameter in `~/.gnupg/scdaemon.conf`. The value '0' refers to the first available serial port reader and a value of '32768' (default) refers to the first USB reader.

GnuPG with pcscd (PCSC Lite)

pcscd(8) is a daemon which handles access to smartcard (SCard API). If GnuPG's scdaemon fails to connect the smartcard directly (e.g. by using its integrated CCID support), it will fallback and try to find a smartcard using the PCSC Lite driver.

To use pscsd install pcsclite and ccid. Then start and/or enable `pcscd.service`. Alternatively start and/or enable `pcscd.socket` to activate the daemon when needed.

Always use pcscd

If you are using any smartcard with an opensc driver (e.g.: ID cards from some countries) you should pay some attention to GnuPG configuration. Out of the box you might receive a message like this when using `gpg --card-status`

gpg: selecting openpgp failed: ec=6.108

By default, scdaemon will try to connect directly to the device. This connection will fail if the reader is being used by another process. For example: the pcscd daemon used by OpenSC. To cope with this situation we should use the same underlying driver as opensc so they can work well together. In order to point scdaemon to use pcscd you should remove `reader-port` from `~/.gnupg/scdaemon.conf`, specify the location to `libpcsclite.so` library and disable ccid so we make sure that we use pcscd:

~/.gnupg/scdaemon.conf

pcsc-driver /usr/lib/libpcsclite.so
card-timeout 5
disable-ccid

Please check scdaemon(1) if you do not use OpenSC.

Shared access with pcscd

GnuPG `scdaemon` is the only popular `pcscd` client that uses `PCSC_SHARE_EXCLUSIVE` flag when connecting to `pcscd`. Other clients like OpenSC PKCS#11 that are used by browsers and programs listed in Electronic identification are using `PCSC_SHARE_SHARED` that allows simultaneous access to single smartcard. `pcscd` will not give exclusive access to smartcard while there are other clients connected. This means that to use GnuPG smartcard features you must before have to

close all your open browser windows or do some other inconvenient operations. Starting from version 2.2.28 LTS and 2.3.0 you can enable shared access by modifying your `scdaemon.conf` file and adding `pcsc-shared` line end of it.

Multi applet smart cards

When using YubiKeys or other multi applet USB dongles with OpenSC PKCS#11 may run into problems where OpenSC switches your Yubikey from OpenPGP to PIV applet, breaking the `scdaemon`.

You can hack around the problem by forcing OpenSC to also use the OpenPGP applet. Open `/etc/opensc.conf` file, search for Yubikey and change the `driver = "PIV-II";` line to `driver = "openpgp";`. If there is no such entry, use `pcsc_scan`. Search for the Answer to Reset `ATR: 12 34 56 78 90 AB CD ...`. Then create a new entry.

/etc/opensc.conf

```
...
card_atr 12:23:34:45:67:89:ab:cd:... {
    name = "YubiKey Neo";
    driver = "openpgp"
}
...
```

After that you can test with `pkcs11-tool -O --login` that the OpenPGP applet is selected by default. Other PKCS#11 clients like browsers may need to be restarted for that change to be applied.

Tips and tricks

Different algorithm

You may want to use stronger algorithms:

~/.gnupg/gpg.conf

...

personal-digest-preferences SHA512
cert-digest-algo SHA512
default-preference-list SHA512 SHA384 SHA256 SHA224 AES256 AES192 AES CAST5 ZLIB BZIP2 ZIP Uncompressed
personal-cipher-preferences TWOFISH CAMELLIA256 AES 3DES

In the latest version of GnuPG, the default algorithms used are SHA256 and AES, both of which are secure enough for most people. However, if you are using a version of GnuPG older than 2.1, or if you want an even higher level of security, then you should follow the above step.

Encrypt a password

It can be useful to encrypt some password, so it will not be written in clear on a configuration file. A good example is your email password.

First create a file with your password. You **need** to leave **one** empty line after the password, otherwise gpg will return an error message when evaluating the file.

Then run:

$ gpg -e -a -r *user-id your_password_file*

`-e` is for encrypt, `-a` for armor (ASCII output), `-r` for recipient user ID.

You will be left with a new `your_password_file.asc` file.

Tip: pass automates this process.

Change trust model

By default GnuPG uses the Web of Trust as the trust model. You can change this to Trust on first use by adding `--trust-model=tofu` when adding a key or adding this option to your GnuPG configuration file. More details are in this email to the GnuPG list.

Hide all recipient id's

By default the recipient's key ID is in the encrypted message. This can be removed at encryption time for a recipient by using `hidden-recipient user-id`. To remove it for all recipients add `throw-keyids` to your configuration file. This helps to hide the receivers of the message and is a limited countermeasure against traffic analysis (i.e. using a little social engineering, anyone who is able to decrypt the message can check whether one of the other recipients is the one they suspect). On the receiving side, it may slow down the decryption process because all available secret keys must be tried (e.g. with `--try-secret-key user-id`).

Using caff for keysigning parties

To allow users to validate keys on the keyservers and in their keyrings (i.e. make sure they are from whom they claim to be), PGP/GPG uses the Web of Trust. Keysigning parties allow users to get together at a physical location to validate keys. The Zimmermann-Sassaman key-signing protocol is a way of making these very effective. Here you will find a how-to article.

For an easier process of signing keys and sending signatures to the owners after a keysigning party, you can use the tool *caff*. It can be installed from the AUR with the package caff-git[AUR].

To send the signatures to their owners you need a working MTA. If you do not have already one, install msmtp.

Always show long ID's and fingerprints

To always show long key ID's add `keyid-format 0xlong` to your configuration file. To always show full fingerprints of keys, add `with-fingerprint` to your configuration file.

Custom capabilities

For further customization also possible to set custom capabilities to your keys. The following capabilities are available:

- Certify (only for master keys) - allows the key to create subkeys, mandatory for master keys.
- Sign - allows the key to create cryptographic signatures that others can verify with the public key.
- Encrypt - allows anyone to encrypt data with the public key, that only the private key can decrypt.
- Authenticate - allows the key to authenticate with various non-GnuPG programs. The key can be used as e.g. an SSH key.

  It is possible to specify the capabilities of the master key, by running:

  $ gpg --full-generate-key --expert

  And select an option that allows you to set your own capabilities.

  Comparably, to specify custom capabilities for subkeys, add the

`--expert` flag to `gpg --edit-key`, see #Edit your key for more information.

Troubleshooting

Not enough random bytes available

When generating a key, gpg can run into this error:

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy!

To check the available entropy, check the kernel parameters:

$ cat /proc/sys/kernel/random/entropy_avail

A healthy Linux system with a lot of entropy available will have return close to the full 4,096 bits of entropy. If the value returned is less than 200, the system is running low on entropy.

To solve it, remember you do not often need to create keys and best just do what the message suggests (e.g. create disk activity, move the mouse, edit the wiki - all will create entropy). If that does not help, check which service is using up the entropy and consider stopping it for the time. If that is no alternative, see Random number generation#Alternatives.

su

When using `pinentry`, you must have the proper permissions of the terminal device (e.g. `/dev/tty1`) in use. However, with *su* (or *sudo*), the ownership stays with the original user, not the new one. This means that pinentry will fail with a `Permission denied` error, even as root. If this happens when attempting to use ssh, an error like `sign_and_send_pubkey: signing failed: agent refused operation` will be returned. The

fix is to change the permissions of the device at some point before the use of pinentry (i.e. using gpg with an agent). If doing gpg as root, simply change the ownership to root right before using gpg:

chown root /dev/ttyN # where N is the current tty

and then change it back after using gpg the first time. The equivalent is true with `/dev/pts/`.

Note: The owner of tty *must* match with the user for which pinentry is running. Being part of the group `tty` **is not** enough.

Tip: If you run gpg with `script` it will use a new tty with the correct ownership:

script -q -c "gpg --gen-key" /dev/null

Agent complains end of file

If the pinentry program is `/usr/bin/pinentry-gnome3`, it needs a DBus session bus to run properly. See General troubleshooting#Session permissions for details.

Alternatively, you can use a variety of different options described in #pinentry.

KGpg configuration permissions

There have been issues with kgpg being able to access the `~/.gnupg/` options. One issue might be a result of a deprecated *options* file, see the bug report.

GNOME on Wayland overrides SSH agent socket

For Wayland sessions, `gnome-session` sets `SSH_AUTH_SOCK` to the standard gnome-keyring socket,

`$XDG_RUNTIME_DIR/keyring/ssh`. This overrides any value set elsewhere.

See GNOME/Keyring#Disable keyring daemon components on how to disable this behavior.

mutt

Mutt might not use *gpg-agent* correctly, you need to set an environment variable `GPG_AGENT_INFO` (the content does not matter) when running mutt. Be also sure to enable password caching correctly, see #Cache passwords.

See this forum thread.

"Lost" keys, upgrading to gnupg version 2.1

When `gpg --list-keys` fails to show keys that used to be there, and applications complain about missing or invalid keys, some keys may not have been migrated to the new format.

Please read GnuPG invalid packet workaround. Basically, it says that there is a bug with keys in the old `pubring.gpg` and `secring.gpg` files, which have now been superseded by the new `pubring.kbx` file and the `private-keys-v1.d/` subdirectory and files. Your missing keys can be recovered with the following commands:

```
$ cd
$ cp -r .gnupg gnupgOLD
$ gpg --export-ownertrust > otrust.txt
$ gpg --import .gnupg/pubring.gpg
$ gpg --import-ownertrust otrust.txt
$ gpg --list-keys
```

gpg hanged for all keyservers (when trying to receive keys)

If gpg hanged with a certain keyserver when trying to receive keys, you might need to kill dirmngr in order to get access to other keyservers which are actually working, otherwise it might keeping hanging for all of them.

Smartcard not detected

Your user might not have the permission to access the smartcard which results in a `card error` to be thrown, even though the card is correctly set up and inserted.

One possible solution is to add a new group `scard` including the users who need access to the smartcard.

Then use udev rules, similar to the following:

/etc/udev/rules.d/71-gnupg-ccid.rules

ACTION=="add", SUBSYSTEM=="usb", ENV{ID_VENDOR_ID}=="1050", ENV{ID_MODEL_ID}=="0116|0111", MODE="660", GROUP="scard"

One needs to adapt VENDOR and MODEL according to the `lsusb` output, the above example is for a YubikeyNEO.

server 'gpg-agent' is older than us (x < y)

This warning appears if `gnupg` is upgraded and the old gpg-agent is still running. Restart the *user*'s `gpg-agent.socket` (i.e., use the `--user` flag when restarting).

IPC connect call failed

Make sure `gpg-agent` and `dirmngr` are not running with `killall gpg-agent dirmngr` and the `$GNUPGHOME/crls.d/` folder has permission set to `700`.

By default, the gnupg package uses the directory `/run/user/$UID/gnupg/` for sockets. GnuPG documentation states this is the preferred directory (not all file systems are supported for sockets). Validate that your `agent-socket` configuration specifies a path that has an appropriate file system. You can find the your path settings for `agent-socket` by running `gpgconf --list-dirs agent-socket`.

Test that `gpg-agent` starts successfully with `gpg-agent --daemon`.

Mitigating Poisoned PGP Certificates

In June 2019, an unknown attacker spammed several high-profile PGP certificates with tens of thousands (or hundreds of thousands) of signatures (CVE-2019-13050) and uploaded these signatures to the SKS keyservers. The existence of these poisoned certificates in a keyring causes gpg to hang with the following message:

gpg: removing stale lockfile (created by 7055)

Possible mitigation involves removing the poisoned certificate as per this blog post.

Invalid IPC response and Inappropriate ioctl for device

The default pinentry program is `/usr/bin/pinentry-gtk-2`. If gtk2 is unavailable, pinentry falls back to `/usr/bin/pinentry-curses` and causes signing to fail:

gpg: signing failed: Inappropriate ioctl for device
gpg: [stdin]: clear-sign failed: Inappropriate ioctl for device

You need to set the GPG_TTY environment variable for the pinentry programs `/usr/bin/pinentry-tty` and `/usr/bin`

`/pinentry-curses`.

$ export GPG_TTY=$(tty)

Keyblock resource does not exist

If you get an error like this when trying to import keys

gpg: keyblock resource '*gnupg_home*/pubring.kbx': No such file or directory

it is because GnuPG will not create its home directory if it does not yet exist. Simply create it manually

$ mkdir -m 700 *gnupg_home*

See also

- GNU Privacy Guard Homepage
- Alan Eliasen's GPG Tutorial
- RFC 4880 — "OpenPGP Message Format"
- gpg.conf recommendations and best practices
- Fedora:Creating GPG Keys
- Debian:Subkeys
- Protecting code integrity with PGP
- A more comprehensive gpg Tutorial
- /r/GPGpractice - a subreddit to practice using GnuPG.